Lab Assignment #2

(ver. 2)

Goals: To gain experience working with C++ structures such as classes, pointers, linked lists, dynamic arrays, etc. To work on an interesting, open-ended problem while practicing programming.

Due Date: Fri, Feb 10 @ 10pm

Warning: If the assignment seems simple at a glance, be aware that it has a number of issues to resolve, and they are not trivial. So start working on it immediately, and pace yourself. If you only start near the end of the period, it will almost certainly remain incomplete. (Even I cannot do it well in 1-2 days, and neither should you expect such feats of yourself.)

Submission: A turn-in link, called "Assignment Submission", is available on the course web site.
The submission mechanism we will use (UW-Catalyst DropBox tool) is the same as for Lab #1 and Hw #1.

Mechanics: You will be working individually for the coding and project description portion of this project assignment. You are, however, free to discuss the assignment with classmates, but without sharing actual code or written descriptions.

Product Requirements:

For this lab assignment, we will perform interesting text processing tasks. The specific problem has two parts, and can be briefly stated as follows:

(A) Given an input text file (without any special formatting), find the total number of words in that file, the total number of different words, and the top-20 most frequently used words (along with their respective numbers), with some exceptions, as described below. Display those results appropriately as follows (the numbers shown are samples only):

```
   Total number of English words:    257
   Total number of different words:   56
   Top most frequent words (and their frequencies):
      day         18
      rain        16
      wind        12
      Seattle      7
      snow         5
      ...         ...
```

etc.

Then, based on the collection of these most frequent words, determine and display what type of information is contained in the file. Possible types include: weather forecast, driving directions, privacy policy, and cooking recipes.

(B) For the case of input files containing driving directions (from point A to point B), reverse those directions and display the results (i.e., the directions how to get from point B to point A).
A sample example is provided below.

See below for a more specific description of the various aspects of the problem.

Expected Inputs:

Input data will come from plain text files, whose names must be given as command-line arguments, along with command-line options indicating the specific type of processing required. For example:

(A) for computing word statistics on a file and classifying it, the command-line input will look like this:
`TextProcess -classify uncategorized.txt non_descript_words.txt category_word_assoc.txt`, where
`uncategorized.txt` is a text file that needs to be counted and classified;
`non_descript_words.txt` is a text file with common English words that can't help with categorizing (see below);
`category_word_assoc.txt` is a text file containing an association of each of three given categories (see below) with specific words that likely imply such categories (e.g., "rain" suggests the "weather forecast" category, while "walnuts" likely implies the "cooking recipe" category).

(B) for reversing driving directions, the command-line input may look like this:
`TextProcess -reverse_dir driving_dir.txt common_dir_terms.txt replace_word_assoc.txt`,

where
`driving_dir.txt` is a text file that contains driving directions between two points (similar to but simpler than what MapQuest or similar online travel direction services offer; see below for a specific example);
`common_dir_terms.txt` is a text file that contains common terms used in describing directions (e.g., "street", "avenue", "turn", "intersection", "continue", "straight", etc.);
`replace_word_assoc.txt` is a text file that contains associations of words to be replaced when reversing directions (e.g., "south" -> "north", "left" -> "right", etc.).

Expected Outputs:
- if command-line option is "`-classify`":
The top-20 most frequently used words in the file (with the exception of some common non-descript words in the English language, e.g., "a", "the", "in", "for", "I", "you", "and", etc.), along with their actual frequency (in absolute numbers). Also, output the total number of words in the file and the total number of different words. Finally, based on the top most frequently used words in the text, classify which one of several predefined categories (e.g., "driving directions", "weather forecast", or "cooking recipe") the input belongs to.

- if command-line option is "`-reverse_dir`":
A list of (reversed) driving directions, from point B to point A. They must read properly in English.

Assumptions and Simplifications:
It is okay to assume without verifying explicitly in your code the following:

(a) Words of the same root but otherwise not identical to each other can be considered different words, for the purposes of this assignment (e.g., "differ", "differing", "different"). However, punctuation needs to be removed from words, so as not to obscure genuinely identical words (e.g., "college" and "college," are indeed the same word).
If you are able to correctly identify almost identical words, this will be considered for extra credit, but be warned that such a task would be technically much more difficult (not in terms of C++ programming, but in terms of the exact algorithm that can be applied).

(b) Whenever your program `TextProcess` is invoked with the option "`-reverse_dir`", the input file can be assumed to contain driving directions in plain text, e.g.:
"*Start at 12345 Main Street, near the college Starbucks. Go north for 2 miles. Then, turn right at the intersection with Inspiration Avenue. Continue along Inspiration Avenue for 3.2 miles, and turn left on the corner of Joy Street. After 0.3 miles on Joy Street, arrive at Barnes & Noble.*"

(c) There is no need to format your output in the case of driving directions. A sequence of sentences in plain text is sufficient. From the sample directions in (b), the output could be approximately this:
"*Start at Barnes & Noble on Joy Street. Go for 0.3 miles, and turn right on the corner of Inspiration Avenue. Continue along Inspiration Avenue for 3.2 miles. Then, turn left at the intersection with Main Street. Go south for 2 miles. Arrive at 12345 Main Street, near the college Starbucks.*"

(d) For reversing driving directions, assume that all streets are bi-directional (i.e., no one-way streets), so that your charted route in both directions will be identical (except, of course, reversed).

Specific Programming-Related Requirements:
- Place all your functions and necessary variables into appropriate classes.
- Use a linked list to store the individual words of the text file, once retrieved from the input file.
- You may use other structures (e.g., dynamic arrays) for sorting the words by frequency of occurrence. Or use other linked lists…
- Functions for reading in one word at a time from a given input text file will be supplied by me, shortly. You will need to use them in your code *without modification*.
- Do not hardcode *any words* (i.e., potential input or output text) in your C++ code. Instead, use separate input text files as indicated on the command line (e.g., "`replace_word_assoc.txt`") from which to retrieve any necessary terms, associations, etc. for your problem domain.

Tips:
- Consider what classes you may need to have in your program in order to model the necessary data and incorporate the needed functions that operate on the data. Which functions will naturally go into which classes? What interfaces (i.e., public methods) would each class have to expose to its clients, and what methods might be better left private?
- Consider what data structures (linked list or dynamic arrays) will be best to use for sorting the input by frequency of occurrences; for keeping the list of common non-descript terms; for keeping the associations for replacing words; for storing landmarks (in the driving directions), which need to remain unchanged in the reverse directions; etc. You may find that you need different types of

structures for each of those specific purposes, depending on what algorithms you plan to use, and how well they are supported by such data structures. Consult the class discussion on the tradeoffs between arrays, linked lists, and files.

- Getting the correct output of directions would involve thinking through which types of directions need to be replaced with which others, and how to reverse parts of the sentences of the original. Experimentation will help you immensely with this. Create at least several different test cases and use them for testing and debugging, to make sure your code performs the reversal correctly (and understandably to a human reader).
  To allow time for the experimentation and associated "tweaking" based on specific test cases, you will need to start working on this project immediately, so that there will be time left for those important experimentations when your code already reaches a state where it almost works.
- You are encouraged to share sample input files among all students; use the discussion board for this – in a special forum for this purpose (under "Lab #2 discussion").
- Feel free to use public domain files as sample input for initial testing. For example, (some of) the works of Shakespeare, Arthur Conan Doyle, etc. can be found on the Internet if you need larger files for testing (which I recommend doing). For smaller files, you can easily create them and/or edit yourselves.
- Start by understanding the domain well. Experiment on paper with sample input files you can think of.
- **Ask questions**; I can answer; talk to your fellow classmates about what questions they have. Certain parts of the product description may be vague to begin with (which is quite common in initial product descriptions, even after a round of conversations), and in need of clarification before you have a firm grasp on what it is exactly that needs to be built.
- We will discuss these and many other issues in our next class meeting(s). The more you have thought about the problem earlier, the more value you will get out of such discussions, and the more valuable I can be in answering questions and guiding you toward effective designs, algorithms, and implementation choices, etc.

Hall of Fame:

I would like to show the best results from this assignment in class in a Hall of Fame demo.
You can nominate your own work for the Hall of Fame, if you wish. A nomination can be based on how well your program reverses directions (including, perhaps, in more tricky cases) and/or based on how well your program classifies different inputs.
If you wish to nominate your work, email me at the time of your final submission of the lab assignment, and I will discuss with you what you would like to showcase, and how it can be accomplished.

Evaluation:

In the evaluation of your project work, I will be looking to see not only that your code compiles and runs correctly with my test cases, but also that you have an associated write-up, in which you have:
(a) described your designs clearly, including classes and their interfaces (public methods), and justified those designs over alternatives you have considered. This includes justifying briefly why you chose linked lists over arrays (or vice versa, as the case may be) for the various situations mentioned above;
(b) created a reasonable (even if incomplete) set of test cases to test your program for functional correctness as well as for robustness against various types of inputs;
(c) described clearly and concisely what you have been able to accomplish of the required tasks, and what not;
(d) listed the bugs you are aware of in your code (note: every code has bugs, and that's okay so long as they are not critical, preventing the main functionality from working);
(e) described what, if any, additional (unspecified here) functionality you were able to build;
(f) shown a snapshot of your program running on a sample input, and producing its output;
(g) listed the number of hours it took you to work on this project, broken down by hours before coding (i.e., designs, clarifying requirements, etc.) and hours for coding and testing;
(h) organized and presented your write-up well.

Deliverables:

I will expect to see the following deliverables by the due date:
(1) A write-up addressing all the questions from the Evaluation section above – this should be clear and concise; extra length is not a virtue. Use a common document format, to make sure that I have the software to open it and read your write-up. Name your write-up file in a way that will be easy for me to identify (e.g., `*YourLastName*-css342-lab2-writeup.doc`, where *`YourLastName`* is your last name).
(2) Your source code – including only your `*.cpp` (C++ source) and `*.h` (C++ header) files. No executables, no object files, no project files are necessary or desired.
(3) Your test data (e.g., at least 3 sample files). Call them "`test1.txt`", "`test2.txt`", and "`test3.txt`".
(4) Your own versions of the various data files. Keep their names as specified: "`non_descript_words.txt`", "`category_word_assoc.txt`", "`common_dir_terms.txt`", and "`replace_word_assoc.txt`".